МИНОБРНАУКИ РОССИИ

Федеральное государственное автономное образовательное учреждения высшего образования

«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт компьютерных технологий и информационной безопасности

Кафедра математического обеспечения и применения ЭВМ

**ЛАБОРАТОРНАЯ РАБОТА № 4**

по дисциплине

**«Объектно-ориентированное программирование»**

на тему:

**«Использование библиотеки STL»**

*Вариант № 4*

Выполнил:

Студент группы

КТбо2-8

Макаров С. В.

Проверил:

Тарасов С. А.

Оценка

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_ 2020 г.

Таганрог 2020

# **1 ФОРМУЛИРОВКА ЗАДАНИЯ**

Класс “ Записная книжка (Notebook)” с полями: ФИО, номер телефона, дата рождения. Вывести по задаваемым полям данные о записи.

# **2 СПЕЦИФИКАЦИЯ КЛАССОВ**

class Date : public IStringConvertable

{

public:

Date();

Date(int day, int month, int year);

virtual std::string ToString() const override;

bool operator==(const Date& other) const;

bool operator<(const Date& other) const;

int GetDay() const;

int GetMonth() const;

int GetYear() const;

int GetDaysCount() const;

static int GetDaysCountInMonth(int month, int year);

static int GetDaysCountInYear(int year);

private:

int \_day;

int \_month;

int \_year;

};

class PersonInitial : public IStringConvertable

{

using string\_arg = const std::string&;

public:

PersonInitial();

PersonInitial(string\_arg name, string\_arg surname, string\_arg patronyimc);

virtual std::string ToString() const override;

std::string GetName() const;

std::string GetSurname() const;

std::string GetPatronymic() const;

private:

std::string \_name;

std::string \_surname;

std::string \_patronyimc;

};

class PhoneNumber : public IStringConvertable

{

using string\_arg = const std::string&;

public:

PhoneNumber();

PhoneNumber(string\_arg counryCode, string\_arg abonentCode, string\_arg number);

virtual std::string ToString() const override;

std::string GetCountyCode() const;

std::string GetAbonentCode() const;

std::string GetNumber() const;

private:

std::string \_counryCode;

std::string \_abonentCode;

std::string \_number;

};

class Note : public IStringConvertable

{

public:

Note();

Note(const PhoneNumber& number, const Date& date, const PersonInitial& initial);

virtual std::string ToString() const override;

Date GetBirthDate() const;

PhoneNumber GetPhoneNumber() const;

PersonInitial GetInitial() const;

private:

PhoneNumber \_number;

Date \_birthDate;

PersonInitial \_initial;

}

class ConsoleInteractor

{

public:

void Run();

private:

std::unique\_ptr<INotesContainer> \_notebook;

template<typename T>

void ReadVar(T& var) const;

bool Init();

int ReadCommand() const;

void TaskReadFileAndBuild();

void TaskPrintNotebook() const;

void TaskSwitchPredicateAndRun() const;

void TaskPrintPredicates() const;

void PrintCommands() const;

void ToUniqueFormat(std::string& destString) const;

};

class IStringConvertable

{

public:

virtual std::string ToString() const = 0;

virtual ~IStringConvertable() {}

};

class INotesContainer : public IStringConvertable

{

public:

virtual void Add(const Note& value) = 0;

virtual std::vector<Note> Find(std::function<bool(const Note&)> predicate) const = 0;

virtual std::vector<Note> FindByKey(const Date& key) const = 0;

virtual std::string ToString() const = 0;

virtual ~INotesContainer()

{}

};

class VectorNotebook : public INotesContainer

{

public:

virtual void Add(const Note& value) override;

virtual std::vector<Note> Find(std::function<bool(const Note&)> predicate) const override;

virtual std::vector<Note> FindByKey(const Date& key) const override;

virtual std::string ToString() const override;

private:

std::vector<Note> \_container;

};

class MapNotebook : public INotesContainer

{

public:

virtual void Add(const Note& value) override;

virtual std::vector<Note> Find(std::function<bool(const Note&)> predicate) const override;

virtual std::vector<Note> FindByKey(const Date& key) const override;

virtual std::string ToString() const override;

private:

std::multimap<Date, Note> \_container;

};

class NotebookBuilder

{

public:

template<typename TContainer>

std::unique\_ptr<INotesContainer> Build() const;

std::unique\_ptr<INotesContainer> BuildFromType(const std::string& typeName) const;

};

template<typename TContainer>

inline std::unique\_ptr<INotesContainer> NotebookBuilder::Build() const

{

if (!std::is\_base\_of<INotesContainer, TContainer>::value)

{

throw std::exception("Invalid type to build");

}

return std::make\_unique<TContainer>(TContainer());

}

class NoteBuilder

{

public:

Note Build(const std::string& data) const;

std::vector<Note> Build(const std::vector<std::string>& data) const;

template<typename T>

T ParseFromStream(std::istream& sin) const;

};

# **3 ИСПОЛЬЗУЕМЫЕ МАТЕМАТИЧЕСКИЕ ЗАВИСИМОСТИ И АЛГОРИТМЫ**

В данной программе используется класс Дата (Date), который представляет дату. Для корректной работы программы необходимо выбирать, в какой месяц сколько дней – 30, 31, 28 или 29. Необходимо учитывать – високосный года или нет.

# **4 ДИАГРАММА КЛАССОВ**

Uml диаграмма классов изображена на рисунке 1.

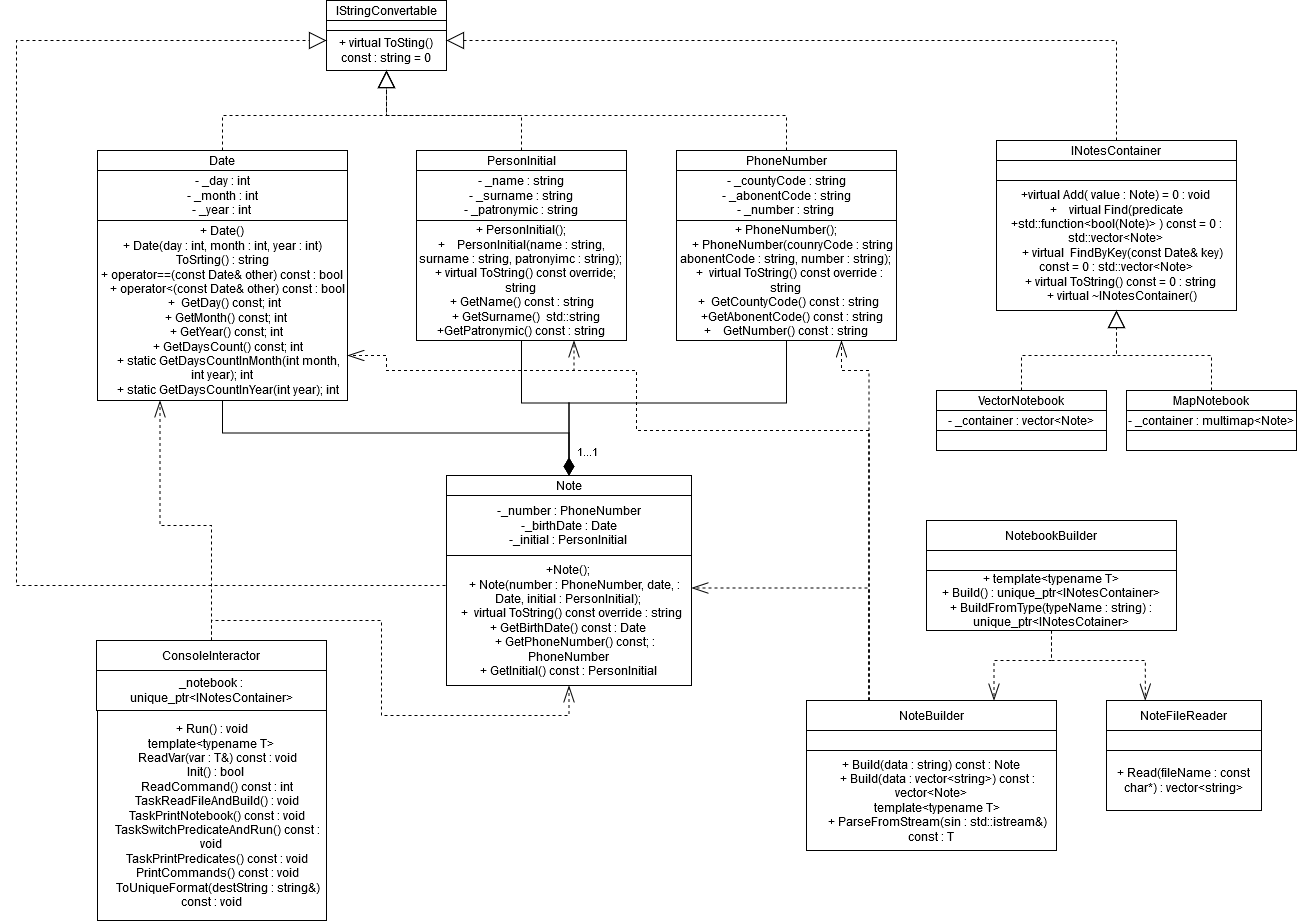


Рисунок 1 – Диаграмма классов

**ЛИСТИНГ ПРОГРАММЫ**

int main()

{

ConsoleInteractor app;

app.Run();

return 0;

}

void ConsoleInteractor::Run()

{

if (!Init())

{

return;

}

PrintCommands();

int command;

while (true)

{

try

{

command = ReadCommand();

switch (command)

{

case 0:

return;

case 1:

TaskReadFileAndBuild();

cout << "success\n";

break;

case 2:

TaskSwitchPredicateAndRun();

break;

case 3:

TaskPrintNotebook();

break;

case 4:

PrintCommands();

break;

default:

cout << "Invalid command\n";

break;

}

}

catch (const std::exception & e)

{

std::cout << e.what() << "\n";

}

}

}

bool ConsoleInteractor::Init()

{

std::string type;

cout << "Enter name of base type for notebook. for example: VECTOR, MULTIMAP, ... or type EXIT to\n";

NotebookBuilder builder;

while (true)

{

try

{

cout << ">>> ";

cin >> type;

ToUniqueFormat(type);

if (type.compare("EXIT") == 0)

{

return false;

}

\_notebook = builder.BuildFromType(type);

return true;

}

catch (const std::exception & err)

{

cout << err.what() << '\n';

cout << "Try again\n";

}

}

}

int ConsoleInteractor::ReadCommand() const

{

int command;

cout << "\nenter command: ";

ReadVar(command);

return command;

}

void ConsoleInteractor::TaskReadFileAndBuild()

{

std::string fileName;

cout << "Enter File Name: ";

cin >> fileName;

NoteFileReader reader;

NoteBuilder builder;

auto readedFile = reader.Read(fileName.c\_str());

auto buildedNotes = builder.Build(readedFile);

for (auto& note : buildedNotes)

{

\_notebook->Add(note);

}

}

void ConsoleInteractor::TaskPrintNotebook() const

{

cout << \_notebook->ToString() << "\n";

}

void ConsoleInteractor::TaskSwitchPredicateAndRun() const

{

TaskPrintPredicates();

int predIndex;

cout << "Enter predicate index (starts from zero): ";

ReadVar(predIndex);

std::vector<Note> finded;

switch (predIndex)

{

case 0:

{

cout << "Enter day: ";

int day;

ReadVar(day);

if (day <= 0 || day > 31)

{

throw std::runtime\_error("Invalid day readed");

}

finded = \_notebook->Find([day](const Note& note) -> bool

{

if (note.GetBirthDate().GetDay() == day)

{

return true;

}

return false;

});

}

break;

case 1:

{

cout << "Enter month: ";

int month;

ReadVar(month);

if (month <= 0 || month > 12)

{

throw std::runtime\_error("Invalid day readed");

}

finded = \_notebook->Find([month](const Note& note) -> bool

{

if (note.GetBirthDate().GetMonth() == month)

{

return true;

}

return false;

});

}

break;

case 2:

{

cout << "Enter date in format dd.mm.yyyy : ";

NoteBuilder builder;

string data;

cin >> data;

stringstream sstr(data);

Date date = builder.ParseFromStream<Date>(sstr);

finded = \_notebook->FindByKey(date);

}

break;

case 3:

{

cout << "Enter name: ";

string name;

cin >> name;

for (auto& letter : name)

{

if (!isalpha(letter))

{

throw std::runtime\_error("Invalid name");

}

letter = tolower(letter);

}

name[0] = toupper(name[0]);

finded = \_notebook->Find([name](const Note& note) -> bool

{

if (note.GetInitial().GetName() == name)

{

return true;

}

return false;

});

}

break;

case 4:

{

cout << "Enter county code: ";

string code;

cin >> code;

for (auto& letter : code)

{

if (!isdigit(letter))

{

throw std::runtime\_error("Invalid county code");

}

}

finded = \_notebook->Find([code](const Note& note) -> bool

{

if (note.GetPhoneNumber().GetCountyCode() == code)

{

return true;

}

return false;

});

}

break;

default:

throw std::runtime\_error("Invalid predicate index");

break;

}

cout << "finded vector contains " << finded.size() << " elements:\n";

for (auto& item : finded)

{

cout << item.ToString() << "\n";

}

cout << "\n";

}

void ConsoleInteractor::TaskPrintPredicates() const

{

cout << "Predicates to test:\n";

cout << "0: find notes with inputed day\n";

cout << "1: find notes with inputed month\n";

cout << "2: find notes with inputed Date\n";

cout << "3: find notes with inputed Name\n";

cout << "4: find notes with inputed phone number's county code\n";

}

void ConsoleInteractor::PrintCommands() const

{

system("cls");

cout << "Commands: 0 - exit\n";

cout << " 1 - read from file and add to container\n";

cout << " 2 - test predicates\n";

cout << " 3 - print notebook\n";

cout << " 4 - clear console\n";

}

void ConsoleInteractor::ToUniqueFormat(std::string& destString) const

{

for (auto& letter : destString)

{

letter = toupper(letter);

}

}

template void ConsoleInteractor::ReadVar<int>(int& var) const;

template<typename T>

void ConsoleInteractor::ReadVar(T& var) const

{

string input;

cin >> input;

istringstream sin(input);

char c;

if (!(sin >> var) || (sin >> c))

{

throw std::runtime\_error("Input error. Failed to enter a variable");

}

}

void MapNotebook::Add(const Note& value)

{

\_container.insert(std::make\_pair(value.GetBirthDate(), value));

}

std::vector<Note> MapNotebook::Find(std::function<bool(const Note&)> predicate) const

{

std::vector<Note> finded;

for (auto& note : \_container)

{

if (predicate(note.second))

{

finded.push\_back(note.second);

}

}

finded.shrink\_to\_fit();

return finded;

}

std::vector<Note> MapNotebook::FindByKey(const Date& key) const

{

auto range = \_container.equal\_range(key);

std::vector<Note> finded;

for (auto i = range.first; i != range.second; ++i)

{

finded.push\_back(i->second);

}

finded.shrink\_to\_fit();

return finded;

}

std::string MapNotebook::ToString() const

{

std::ostringstream sstr;

for (auto& item : \_container)

{

sstr << item.second.ToString() << "\n";

}

return sstr.str();

}

std::unique\_ptr<INotesContainer> NotebookBuilder::BuildFromType(const std::string& typeName) const

{

std::unique\_ptr<INotesContainer> notebook;

if (typeName.compare("VECTOR") == 0)

{

notebook = Build<VectorNotebook>();

}

else if (typeName.compare("MULTIMAP") == 0)

{

notebook = Build<MapNotebook>();

}

else

{

throw std::runtime\_error("Invalid type to create notebook");

}

return notebook;

}

template<>

inline Date NoteBuilder::ParseFromStream<Date>(std::istream& sin) const

{

int date[3]; // day, month, year

char dot;

for (int i = 0; i < 2; i++)

{

if (!(sin >> date[i]) || !(sin >> dot))

{

throw std::runtime\_error("Parsing data from stream error");

}

}

if (!(sin >> date[2]))

{

throw std::runtime\_error("Parsing data from stream error");

}

if (date[1] <= 0 || date[1] > 12

|| date[2] <= 1900 || date[2] > 3000)

{

throw std::invalid\_argument("Invalid data from stream");

}

int lowerBound = 1, upperBound = Date::GetDaysCountInMonth(date[1], date[2]);

int month = date[1];

if (date[0] < lowerBound || date[0] > upperBound)

{

throw std::invalid\_argument("Invalid data from stream");

}

return Date(date[0], date[1], date[2]);

}

template<>

inline PhoneNumber NoteBuilder::ParseFromStream<PhoneNumber>(std::istream& sin) const

{

std::string numbers[3];

for (int i = 0; i < 3; i++)

{

if (!(sin >> numbers[i]))

{

throw std::runtime\_error("Parsing data from stream error");

}

}

for (int i = 0; i < 3; i++)

{

for (auto& letter : numbers[i])

{

if (!isdigit(letter))

{

throw std::invalid\_argument("Invalid data from stream");

}

}

}

return PhoneNumber(numbers[0], numbers[1], numbers[2]);

}

template<>

inline PersonInitial NoteBuilder::ParseFromStream<PersonInitial>(std::istream& sin) const

{

std::string names[3];

for (int i = 0; i < 3; i++)

{

if (!(sin >> names[i]))

{

throw std::runtime\_error("Parsing data from stream error");

}

}

for (int i = 0; i < 3; i++)

{

for (auto& letter : names[i])

{

if (!isalpha(letter))

{

throw std::invalid\_argument("Invalid data from stream");

}

}

}

for (int i = 0; i < 3; i++)

{

for (auto& letter : names[i])

{

letter = tolower(letter);

}

names[i][0] = toupper(names[i][0]);

}

return PersonInitial(names[0], names[1], names[2]);

}

Note NoteBuilder::Build(const std::string& data) const

{

std::istringstream sin(data);

auto date = ParseFromStream<Date>(sin);

auto number = ParseFromStream<PhoneNumber>(sin);

auto initial = ParseFromStream<PersonInitial>(sin);

return Note(number, date, initial);

}

std::vector<Note> NoteBuilder::Build(const std::vector<std::string>& data) const

{

std::vector<Note> notes;

for (auto& item : data)

{

try

{

notes.push\_back(Build(item));

}

catch (...)

{

}

}

notes.shrink\_to\_fit();

return notes;

}

std::vector<std::string> NoteFileReader::Read(const char\* fileName)

{

std::fstream file;

file.open(fileName, std::ios\_base::in);

if (!file.good())

{

throw std::runtime\_error("Error when opening file");

}

std::vector<std::string> lines;

std::string line;

while (std::getline(file, line))

{

lines.push\_back(std::move(line));

}

lines.shrink\_to\_fit();

return lines;

}

void VectorNotebook::Add(const Note& value)

{

\_container.push\_back(value);

}

std::vector<Note> VectorNotebook::Find(std::function<bool(const Note&)> predicate) const

{

std::vector<Note> finded;

for (auto& note : \_container)

{

if (predicate(note))

{

finded.push\_back(note);

}

}

finded.shrink\_to\_fit();

return finded;

}

std::vector<Note> VectorNotebook::FindByKey(const Date& key) const

{

auto predicate = [key](const Note& note) -> bool

{

if (note.GetBirthDate() == key)

{

return true;

}

return false;

};

return Find(predicate);

}

std::string VectorNotebook::ToString() const

{

std::ostringstream sstr;

for (auto& item : \_container)

{

sstr << item.ToString() << "\n";

}

return sstr.str();

}

Date::Date(int day, int month, int year)

: \_day(day),

\_month(month),

\_year(year)

{}

std::string Date::ToString() const

{

std::ostringstream sstr;

sstr << std::setfill('0') << std::setw(2) <<\_day << '.'

<< std::setfill('0') << std::setw(2) << \_month << '.'

<< std::setw(4) << \_year;

return sstr.str();

}

bool Date::operator==(const Date& other) const

{

if (\_day == other.\_day

&& \_month == other.\_month

&& \_year == other.\_year)

{

return true;

}

return false;

}

bool Date::operator<(const Date& other) const

{

int date1 = GetDaysCount();

int date2 = other.GetDaysCount();

if (date1 < date2)

{

return true;

}

return false;

}

int Date::GetDay() const

{

return \_day;

}

int Date::GetMonth() const

{

return \_month;

}

int Date::GetYear() const

{

return \_year;

}

int Date::GetDaysCount() const

{

int res = \_day;

for (int i = 1; i < \_month; i++)

{

res += GetDaysCountInMonth(\_month, \_year);

}

int leapYearsCount = \_year / 4;

int yearsCount = \_year - leapYearsCount;

res += leapYearsCount \* 366 + yearsCount \* 365;

return res;

}

int Date::GetDaysCountInMonth(int month, int year)

{

if (month == 4 || month == 6 || month == 9 || month == 11)

{

return 30;

}

else if (month == 2)

{

if (GetDaysCountInYear(year) == 365)

{

return 28;

}

else

{

return 29;

}

}

else

{

return 31;

}

}

int Date::GetDaysCountInYear(int year)

{

if (year % 4 != 0 || year % 100 == 0 && year % 400 != 0)

{

return 365;

}

return 366;

}

Date::Date()

: \_day(1),

\_month(1),

\_year(2000)

{}

std::string Note::ToString() const

{

std::ostringstream sstr;

sstr << "| ";

sstr << std::setw(35) << std::left << \_initial.ToString() << " | "

<< std::setw(12) << std::right <<\_birthDate.ToString() << " | "

<< std::setw(20) << std::right <<\_number.ToString() << " | ";

return sstr.str();

}

Date Note::GetBirthDate() const

{

return \_birthDate;

}

PhoneNumber Note::GetPhoneNumber() const

{

return \_number;

}

PersonInitial Note::GetInitial() const

{

return \_initial;

}

Note::Note()

: \_number(),

\_birthDate(),

\_initial()

{}

Note::Note(const PhoneNumber& number, const Date& date, const PersonInitial& initial)

: \_number(number),

\_birthDate(date),

\_initial(initial)

{}

PersonInitial::PersonInitial()

: \_name("Ivan"),

\_surname("Ivanov"),

\_patronyimc("Ivanovich")

{}

PersonInitial::PersonInitial(string\_arg name, string\_arg surname, string\_arg patronyimc)

: \_name(name),

\_surname(surname),

\_patronyimc(patronyimc)

{}

std::string PersonInitial::ToString() const

{

std::ostringstream sstr;

sstr << \_surname << ' ' << \_name << ' ' << \_patronyimc;

return sstr.str();

}

std::string PersonInitial::GetName() const

{

return \_name;

}

std::string PersonInitial::GetSurname() const

{

return \_surname;

}

std::string PersonInitial::GetPatronymic() const

{

return \_patronyimc;

}

PhoneNumber::PhoneNumber()

: \_counryCode("8"),

\_abonentCode("900"),

\_number("0000000")

{}

PhoneNumber::PhoneNumber(string\_arg counryCode, string\_arg abonentCode, string\_arg number)

: \_counryCode(counryCode),

\_abonentCode(abonentCode),

\_number(number)

{}

std::string PhoneNumber::ToString() const

{

std::ostringstream sstr;

sstr << '+' << \_counryCode << '-' << \_abonentCode << '-' << \_number;

return sstr.str();

}

std::string PhoneNumber::GetCountyCode() const

{

return \_counryCode;

}

std::string PhoneNumber::GetAbonentCode() const

{

return \_abonentCode;

}

std::string PhoneNumber::GetNumber() const

{

return \_number;

}